Anonymous Traceback for End-to-End Encryption

Erin Kenney[1(✉)], Qiang Tang[2], and Chase Wu[1]

[1] New Jersey Institute of Technology, Newark, NJ 07102, USA
clk8@njit.edu
[2] The University of Sydney, Sydney, NSW 2006, Australia

Abstract. As secure messaging services become ubiquitous, the need for moderation tools that can function within these systems without defeating their purpose becomes more and more pressing. There are several solutions to deal with moderation on a local level, handling harassment and personal-scale issues, but handling wider-scale issues like disinformation campaigns narrows the field; traceback systems are designed for this, but most are incompatible with anonymity.

In this paper, we present Anonymous Traceback, a traceback system capable of functioning within anonymous secure messaging systems. We carefully model security properties, provide two provably secure and simple constructions, with the most practical construction able to preserve anonymity for all but the original source of a reported abusive message. Our implementation shows integration to messaging systems such as Signal is feasible, with client-side overhead smaller than Signals' sealed sender system, and low overhead overall.

1 Introduction

End-to-End Encrypted (E2EE) messaging services have become ubiquitous. People want to claw back some privacy from the nature of the internet, to communicate with actual peace of mind. However, they also open up many problems, a key one being: how do we moderate a messaging system where all messages are hidden? Message Franking [3,5] for instance, allows a receiver of an abusive message to report it and prove the identity of the one who sent it to them. In many cases, just blocking the immediate sender may be insufficient; attackers may abuse E2EE systems through viral messages and misinformation campaigns, which have become a major issue on various online platforms. Such campaigns have spread disinformation about vaccines [1,2], attempted to interfere with elections [14], and even resulted in deaths [9]. To deal with those issues, [13] initiated the study of source tracing in E2EE systems, which enables the server (with the assistance of a reporting client) to follow the path of a forwarded message through the network and locate the original sender of a reported abusive message. A core challenge of traceback is to maintain security of messages as much as possible until a valid report is made to the tracing server.

However, traceback in its original form has several limitations. On one hand, [13] focused on preserving message confidentiality without considering

V. Atluri et al. (Eds.): ESORICS 2022, LNCS 13555, pp. 42–62, 2022.
https://doi.org/10.1007/978-3-031-17146-8_3

anonymity. Their first "path traceback" construction reveals the identities of the whole forwarding chain, while their second "tree traceback" scheme further reveals every user involved with the message. Since the purpose is to reduce the spread of viral misinformation while maintaining user security/privacy as much as possible, identifying much more information beyond the originator of the reported message should be unnecessary. In addition, it is incompatible with (partially) anonymous E2EE messaging systems such as Signal; for the tracing algorithms in [13] to work, it is necessary that the platform logs who is talking to who. Attackers could still abuse an anonymous E2EE system, to spread viral misinformation where traceback cannot be applied.

1.1 Our Contributions

For those reasons, we introduce Anonymous Traceback. We carefully formulate and define the problem and give 2 constructions capable of maintaining the useful core feature of locating the source of a reported (potentially abusive) message, while working within a partially or even fully anonymous messaging system.

Formulating Anonymous Traceback. We take the basic structure of traceback, and rework it to function in an anonymous system. While the general structure of algorithms remains similar to the original traceback [13], several subtleties need to be taken care of, since tracing the source while preserving anonymity may seem to be antagonistic goals. We first alter the existing security property of trace unforgeability to *anonymous trace unforgeability* to account for the larger scope of actions the adversary can take, and introduce new notions for anonymity. Specifically, *pre-trace anonymity* which keeps a user's identity hidden until a trace is made on a message they are involved with, and *post-trace anonymity* which lasts even after a trace is made. We afford originators and forwarders different levels of anonymity as is feasible.

Constructing Anonymous Traceback. We design two constructions, stressing on simplicity for minimal overhead, and compatibility to existing systems.

The first is "*anonymous path traceback*". As we mentioned above, the path traceback in [13] requires explicitly the identity information of each user for the tracing algorithm. Simply encrypting the information renders it trivial to frame innocent users, as the server cannot verify that information during tracing; some means of ensuring the accuracy of the hidden information is necessary. This somewhat naturally leads towards tools such as zero knowledge proofs, which still incur significant overhead. While those would work in theory, we prefer to create a more efficient solution with minimal overhead. We first observe that we can take advantage of the message recipient's position to place responsibility for verifying critical information on them; if the receiver fails to properly check this information, we consider them corrupted and a valid result for tracing.

There is one more subtle threat: that an adversary may try to launch a "delayed replay attack": the adversary sends an abusive message and corrupts one user A in the forwarding chain. A receives the message from a forwarder B, and then later sends a copy of that message to himself pretending B is the

original sender. This is possible due to limited storage space putting a lifespan on previous send information. We observe that the conventional wisdom of adding timestamps resolves this issue. Here we do not need precise synchronization of all the clocks, just roughly close.

Although anonymous path traceback achieves *pre-tracing anonymity*, it cannot manage *post-tracing anonymity*. In an anonymous messaging system, revealing the whole tracing path *also reveals second and more order connections*, which can be used to identify social groups on the platform. This is fairly troublesome: targeted advertising can be considered invasive, and far worse are the possibilities of a hate group identifying a community to target, or a government cracking down on dissent. Even if you trust the messaging platform, that does not mean that data could not end up in the wrong hands regardless.

To preserve more anonymity, we next construct *anonymous source traceback*, capable of retaining anonymity for *all users except the original source* (or an intentional malicious forwarder). However, current tracing method is a step-by-step process, each message's tracing information pointing backwards to the previous message in the forwarding chain. At each step the tracing information is verified, including the sender's identity, meaning the entire path becomes known in the process of finding the original source. To escape this limitation, we observe that the identity information needs to be decoupled from the tracing information. We realize this by separating the platform into a message server and a tracing server. The former is only in charge of regular messaging functionality and maintains identity-related metadata (still preserves anonymity), while the latter stores relevant tracing information. During tracing, the two servers could jointly "emulate" the step-by-step tracing process as before, until an identity is considered as output of the tracing algorithm. We note this two-server architecture is comparable to other work in the literature, and could have several plausible instantiations in practice, see Sect. 4 for more discussion.

Implementation and Evaluations. We implement and benchmark our constructions' cryptographic components and compare them to benchmarks of libsignal, showing that our computational overhead is practical to integrate into a large-scale messaging application - and is in fact less overhead than Signal's sealed sender system currently uses. While it would be additional overhead on top of sealed sender, that still indicates it is low enough to work at scale. In addition, we also provide an estimate of necessary storage space for tracing data compared to the original traceback paper's metrics, which while larger also appears practical.

1.2 Related Work

There have been an increasing number of studies into accountability in encrypted messaging recently. The moderation tool that began this trend is Message Franking [3,5], which is the baseline tool many others, including traceback, expand upon. To the best of our knowledge, the original traceback [13] is the first instance of a system capable of following a message's forwarding trail to find the original source. In addition to the path traceback capability we replicate here in anonymous path

Table 1. Comparison of tracing systems. △, ◬, and ▲ represent degrees of pre-trace anonymity, while ● represents full post-trace anonymity. Hecate does not require the moderator and platform to be separate and non-colluding, but has a similar issue as FACTS when a user is out of tokens otherwise.

	Original [13]	[8]	FACTS [7]	Hecate [6]	Anon path	Anon source
Traces	Path/Tree	Source	Source	Source	Path	Source
Origin Anon.	N/A	N/A	△	◬	▲	▲
Forwarder Anon.	N/A	N/A	△	●	▲	●
Deniability	Likely	Yes	Likely	Yes	No	No
Server count	1	1	1	2*	1	2
Storage Reqs	Moderate	Small	Small	Small	Large	Large

traceback, their second construction, tree traceback, could recover the entire forwarding tree branching out from the initial source.

Peale, Eskandarian, and Boneh [8] also develop a tracing method they call Source Tracking, which improves upon traceback with two constructions capable of tracing to the original sender without needing to store information for each message on the server, or revealing forwarders. However, the more efficient of these constructions loses the property they call "tree unlinkability", which corresponds to some of the original traceback's *user trace confidentiality* property, and both are incompatible with any form of anonymity, as they require senders to identify themselves to the platform.

FACTS [7] improves upon traceback both by finding only the original source, as in our latter construction and source tracking, and also with threshold reporting; FACTS will only allow a message to be traced after it has been reported by a sufficiently large number of users. This is a step towards preventing the abuse of tracing systems by focusing them on the large-scale misinformation they are meant for. However, FACTS is not fully compatible with anonymous messaging systems either. While FACTS could theoretically function on top of sealed sender, it requires that senders request a token and signature from the server prior to sending their message, making anonymity guarantees fragile at best.

Hecate [6] aims to provide tracing within an anonymous system as we do, though they make different trade-offs. Hecate is derived from Asymmetric Message Franking [11] and so achieves deniability where we do not. However, they achieve tracing through each user generating tokens ahead of time with the moderator, one token consumed with each message sent makes them identifiable to the moderator when reported. This carries a key issue: the tokens cannot be generated directly before use or the anonymity guarantees are significantly weakened, similar to [7]. This means users who leave their phones off, or simply send large volumes of messages at once, are not getting the full anonymity guarantee (Table 1).

Orca [12] also works towards accountability in metadata private messaging systems, approached from the opposite end; where we focus on identifying the original source of forwarded messages, ORCA focuses on more personal-scale issues, enabling users to block people harassing them directly without needing

moderator intervention. Orca is a complement to our own work; it allows users to stop harassment directly without needing the intermediary of the platform, while keeping the platform as a whole clean is left to systems like ours.

Ethics of Source Tracing. The ethics of moderation systems will always be a tricky topic. If used without care they can be abused to harass a user-base or silence criticism, and revealing anonymous identities could lead to legal consequences. At the same time, they have undeniable utility in dealing with toxic behavior. We recommend extensive care in using these systems, and would generally not recommend using our first construction; while it is an important theoretical stepping stone, it reveals more information than we are comfortable with. Further discussion on ethics is present in the full version [4].

2 Definitions and Security Models

Our goal is to create a variant of the traceback system that can function within an Anonymous messaging system and avoid compromising that Anonymity.

Cryptographic Primitives. We will use standard cryptographic tools including symmetric key encryption, hash functions, digital signatures and collision resistant pseudorandom functions (which is a pseudorandom function with an extra collision resistance property taking both the key and PRF input as function input, see [13]). The collision resistant PRF function will be referred to as F, but we omit further details due to the page limit.

The Messaging System. For the most part we treat the underlying E2EE Messaging system our systems are built on top of as a black box, but there are a few key assumptions we make about that system:

- There exists a PKI (Public Key Infrastructure) system in place, such that each user U has an associated public key and private key, as well as a certificate $cert_U$ binding their public key to their identity. We make use of these long-term keys as $LTPK_U$ and $LTSK_U$, to sign and verify our primary digital signature and also as an identifier.
- That for anonymous messages, analogous to Signal's Sealed Sender, the sender includes in their encrypted message a certificate $cert_{U_s}$ for recipient to verify their identity, we input it to our RecMsg function in our first two constructions as a confirmation that we and the underlying platform are in agreement on the identity of the sender.

Notations in Anonymous Traceback. Now the following will explain notation for various elements within the constructions:

- **Users** are denoted by U_i, where i is an identification number unique to that user. U_s and U_r indicate the user sending and receiving a message respectively.
- mid is the **message id** value, it is generated by the PRF and used as a key to store trace information in the platform's database.

- tt_s, tt_r, and tt_p represent the **tracing tags** for the sender, receiver, and platform respectively. They contain each party's view of the trace information. When moving to the split server construction, tt_p is split into tt_{ms} and tt_{ts} for the message server and tracing server respectively.
- C_K, C_{PK}, C_{sig} are the **ciphertexts** containing a **tracing key**, **long-term public key**, and **signature** respectively.
- k_i represents the (symmetric) tracing key for the i'th message in a trace, and $\tilde{k_i}$ is the hashed version, to use as the PRF key.
- DB is the **database** of path tracing information, stored as a key: value system with each entry structured as $(mid : C_K, C_{PK}, C_{sig}, ts)$.
- pk_{eph} and sk_{eph} are **asymmetric keys** generated for an **ephemeral signature** used in the second construction, with sig_{eph} representing that signature.
- MDB and TDB are the two halves of the DB **database**, used when the servers are split for the Message Server and Tracing Server respectively. MDB entries are structured $(mid : C_{PK}, C_{sig}, ts, pk_{eph}, sig_{eph})$ and TDB entries are structured $(mid : C_k, pk_{eph})$.

2.1 Anonymous Traceback Syntax

The basic structure of an anonymous traceback scheme is similar to that of the original traceback schemes [13], consisting of the following components:

TagGen$(U_s, m, td_g) \rightarrow k, tt_s$**:** This algorithm generates the tracing key k and the tracing tag tt_s. The input td_g is a catch-all for the data relevant to tracing necessary for tag generation, which varies by construction but always includes the previous tracing key k_{prev} if available. When delivering the message, k is encrypted alongside the message plaintext m for the recipient to access. Compared to the original, we remove the requirement of providing the recipient's identity as we want to keep as few identities involved in each operation as possible.

Svr-Process$(st_{plat}, tt_s, U_r) \rightarrow (mid, tt_p), tt_r$**:** This protocol is used to verify incoming tracing information from a sender (tt_s) prior to delivering the message and logs information into the platform's state st_{plat}. The output comes in two parts: first the Message ID mid and platform tracing information tt_p, typically used to update the database(s) with mid as the key pointing to tt_p, and second the tracing information that is passed on to the recipient, tt_r, which they use for their own verification in **RecMsg**.

RecMsg$(k, U_s, U_r, m, tt_r) \rightarrow td_r$**:** The recipient runs this algorithm to verify the tag tt_r they receive along with the message m, prior to accepting the message. The output catch-all td_r includes cryptographic data identifying the message, for use in submitting a tracing request later. Usually, for **RecMsg** td is simply k. In our constructions, rather than just the identity U_s, we typically use that user's PKI certificate, as is delivered with messages in Sealed Sender.

Svr-Trace$(st_{plat}, m, k, td_t) \rightarrow trace$**:** The protocol that performs the actual tracing operations, utilizing a message m, associated key k, the platform's

state and additional data represented as td_t. The result, *trace*, varies by construction in what information it reveals, but the minimum requirement is enough information to penalize the original source of the message.

One change worth noting is that we remove the **NewMsg** algorithm here, combining its functionality into **TagGen** for readability; while useful as an abstraction in security games, in practice it is only called immediately prior to calling **TagGen** with all of its output. Having **TagGen** check the *td* for a previous key and generate one if absent simplifies things. **Svr-Process** and **Svr-Trace** are now protocols to be executed by different components of a platform working in concert. We take full advantage of this in our latter two constructions by splitting the server into two halves, a Message Server and Tracing Server, each with their own database.

2.2 Security Model

Our overall goals are to maintain the ability of tracing to find the original source of the message, and at the same time preserve as much anonymity as possible. This leads to the following properties.

Anonymity. Retaining anonymity despite the presence of a tracing system is the primary goal of this paper, so we aim to make these definitions as strong as possible. Origin Anonymity defines anonymity for the original source of a specific message, while Forwarder Anonymity defines anonymity for those who forward a different message. Due to lack of space, we briefly explain the high-level intuitions, detailed security games can be found in the full version [4], represented in the $OriginAnon$ and $ForwarderAnon$ games respectively. In addition, our constructions achieve different degrees of anonymity for these two groups, represented by changing oracles: *post-trace anonymity* is the preferred result, where even after a message is traced identities remain hidden, while *pre-trace anoynmity* maintains anonymity until a trace is made (more explanations below).

We give the adversary complete control outside of the context of the challenge message; able to cause message sends, traces, and read from the database both before and after the challenge message is sent. For the challenge, the adversary chooses the message sent, the recipient user, and in $ForwarderAnon$ they even choose the initial sender of the message. Only at the time of message delivery must the two possible senders/forwarders and the recipient be honest, and otherwise the adversary has free reign to corrupt. This also means our anonymity definitions have forward and backward security; unlimited corruptions are allowed both before and after the challenge message, it is only tracing the message that is disallowed, and only for pre-trace anonymity.

They are formulated by allowing the adversary access to multiple oracles: **Send**, to send or forward messages, **Trace**, to perform traces, and **DB** to query the server's database. The primary differences between pre-trace and post-trace anonymity are that for pre-trace anonymity the challenge message and its forwards are flagged so the tracing oracle will refuse to trace them, while post-trace

anonymity oracles have no such restriction. In addition, for anonymous source traceback the **DB** oracle gives access to only the message server's database, since the two servers cannot collude. We choose the message server's database as it is the more useful of the two, containing the encrypted identity information.

It is worth noting that these properties apply on a message-by-message basis. As none of the tracing information (aside from the signature, which is encrypted) derives from identity or is re-used, an originator whose message is traced in a system that guarantees only pre-trace anoynmity loses anonymity for that specific message, but any other messages they have sent or will send in the future remain anonymous until and unless those specific messages are traced.

Limitations of Sender Anonymity. We note that there is an inherent limitation to sender anonymity. As the recipient's identity is known, a curious server could choose to log each recipient and associate them to the message IDs that they receive. This may reveal the identity of the 'first' recipient of each message that is traced, the one who received it from the originator. This leakage cannot be avoided and so excluded in our models.

However, should the system be built on top of a messaging service that has both sender and receiver anonymity, this leak would disappear. No modification of our constructions is needed, as we at no point require the recipient's identity. While there is a decent amount of evidence that Signal would not take advantage of this leak [10], the same cannot be said of all messaging services. In the long run it would be best to build anonymous traceback on top of systems with stronger anonymity guarantees to avoid this leak.

Anonymous Trace Unforgeability. We borrow the basic structure from trace unforgeability in [13], but careful modifications are needed to accommodate the new complications of introducing *anonymity*, and the adversary's additional capabilities. This property ensures that when a trace is performed, an honest user who (a) is not the original source of the message and (b) did not deliberately partition the trace, cannot be framed. We consider the adversary here to be any group of colluding users; as the point of tracing functionality is to assist the platform in moderation we assume the platform will follow our algorithm. In addition, if the platform wants to punish a user they have the authority to do so regardless of a tracing result. Due to page limit, we explain the high-level idea here, for detailed security games, we refer to the full version [4].

Overall, the adversary is allowed to create a database state through the oracles available to them, and then perform a trace of their choosing. They succeed if any of four (two in the case of anonymous source traceback) possible failure conditions arise in that trace: Either a completely empty trace result, an honest user is misidentified as the original source, the reporter never received the message they are reporting, or an honest user is misidentified as a forwarder.

The **Send** and **SendMal** oracles allow causing honest and dishonest users to send messages, respectively. **SendMal** in particular has been expanded upon, and allows further deviation from the protocol and usage of adversarially chosen

values where **Send** behaves honestly. The **NewMsg** oracle allows honestly generating information for a new 'original' message, and is required by the honest **Send** oracle when authoring a new message.

We also add the **ClearDB** oracle to allow an attacker to simulate waiting for the database sliding window to advance, clearing the trace data, to allow attempting delayed replay attacks.

As the adversary succeeds when the game returns true, the advantage expression for *anonymous trace unforgeability* given a specific construction TB is:

$$\mathbf{Adv}_{TB}^{AnonTrUNF}(\mathcal{A}) = Pr[AnonTrUNF_{TB}^{A} \Rightarrow true]$$

We remark that *Trace Partitioning* is a potential attack where a malicious forwarder purposefully breaks the tracing information to appear to be the original sender themselves. Similar to the original [13] we cannot actually prevent this from happening; even if we were to include text matching software to convert copy & paste into forwards (which likely should be done to prevent the non-tech-savvy from accidentally partitioning a trace by missing the forward button or another similar misunderstanding), this would not stop those who modify their client software from having the capability to break the path and become a 'new' originator. Still, even if someone does this, the end result of a trace will still be a bad actor: the one who deliberately broke the path to the originator. Regardless of which occurs, a single bad actor will be removed from the platform, so while imperfect this still allows traceback to fulfill its purpose.

Trace Confidentiality. A property we carry over from [13] that aims to keep message path information hidden from both the user and the platform until and unless a trace is performed. It can be split into two separate properties: *platform trace confidentiality* and *user trace confidentiality*, defining the ability to hide information from the Platform (prior to tracing) and Users respectively.

The goal is simply to keep information about a message's path hidden unless that message is being traced. In practice this comes down to ensuring that from both the user's and platform's view it is impossible to tell whether a given message is a forward or new; if it is not even known whether a message is a forward to begin with, it is of course impossible to determine information such as previous forwarders or the original sender.

We inherit this property from [13] largely unchanged, including the security games; the adversary is given access to tracing information appropriate to either the platform or a receiving user, and succeeds if able to distinguish between a newly authored message (or random string) and a forwarded message.

3 Warm-Up: Anonymous Path Traceback

The original traceback [13] algorithm has a fairly simple core. When messages are sent, a message ID (*mid*) is generated via the PRF F using a freshly generated symmetric key, k. When delivering a message the server uses *mid* as a key to

store tracing metadata: sender and receiver identities and an encryption of the previous key if forwarded (garbage if an originator). Later a report can be made to the server by sending the plaintext and key, used to generate a *mid* value. This *mid* is looked up, the previous key decrypted, used to generate a new *mid*, and so on. When a lookup fails, a garbage previous key was used, and so the message's originator has been found.

Attempting to introduce anonymity to this system quickly creates issues, however. The original system relies heavily on communications being authenticated; should the sender not authenticate to the server, there is no way to ensure that the information on the sender is correct. The need to authenticate anonymously reminds us of anonymous credentials, but while that would work fine to verify that the sender is "allowed to message", it would not ensure that the identity could be accurately revealed later during the tracing procedure. Adding a zero knowledge proof to be verified by the platform could ensure a message would not be sent without a guarantee that a valid signature existed matching the identity that will be recovered later. However, this solution had a significant drawback: compared to secure messaging systems that support a massive scale of users, it was significantly less efficient. We strive for an extremely *simple* solution with minimal overhead.

A key observation of the trust model used for traceback provided our solution - the idea of partitioning a trace. In traceback, the linked-list nature of the tracing information means that a dishonest user can break the connection to previous messages at any time, essentially choosing to become the new "original source" and take responsibility for a message they forward. This is referred to as "partitioning the trace", and is an inherent part of traceback's security model; they cannot be stopped from taking the blame if they choose but either way a bad actor is detected. This ties in nicely with one of the properties of an anonymous communication system: while the server has no information on the sender's identity, the receiver knows it. Therefore, we may pass the responsibility (which is lightweight) to check that the tracing information corresponds to the correct person to the recipient.

The end result of this is our addition of a digital signature. We assume the underlying messaging system already makes use of a PKI, and so use the existing long term keys to sign the tracing information to be certain the sender's identity is accurate. We sign on *mid*, the encryption of the previous tracing key from original traceback (C_K), as well as an encryption of the public key (C_{PK}) used both to verify the signature and identify the sender. Passing the signature along in the clear creates a problem of its own: the server could brute-force attempt to verify the signature with all known public keys to identify the sender, so it is encrypted as well, to be verified by the server only during tracing. Meanwhile, the recipient is expected to verify the signature, and reject messages if they fail. If they accept a bad signature, when it fails to verify the server will know the recipient accepted an untraceable message.

However, there is one subtle issue remaining. While in theory we can treat the database of tracing information as infinite, in practice that is unsustainable. [13] recommends using a sliding window, where database entries are removed after

a certain period of time. However, that leaves recipients with valid tags and signatures that will no longer be rejected for duplicate *mid*. These could be used to recreate that message without the previous entries, framing the sender as the originator of the message's second life even if they only forwarded the original. Thankfully, the solution to this Delayed Replay Attack could be again simple: by adding a timestamp to the values signed on and checking for recentness when the server delivers a message, this exploit is closed off.

3.1 Construction Details

We follow the same general structure here as the original path traceback construction [13], creating a database of trace information keyed by a unique message id value generated via a collision resistant PRF.

TagGen$(U_s, m, k_{prev}, LTSK_{U_s}) \rightarrow (k_i, tt_s, sig)$**:**

1. Randomly generate a new tracing key k_i. If no k_{prev} is provided, also generate a false previous key, for a new message.
2. Generate the timestamp ts.
3. Use a hash to generate $\tilde{k_i}$ from k_i, to separate the key used for the PRF F from the key used in encryption.
4. Calculate the message ID, *mid*, as $\mathsf{F}_{\tilde{k_i}}(m)$.
5. Using k_i, encrypt k_{prev} and the sender's long term public key $LTPK_{U_s}$ as C_K and C_{PK} respectively.
6. Using $LTSK_{U_s}$, generate a digital signature, *sig*, on the combined information (mid, C_K, C_{PK}, ts).
7. Encrypt *sig* using k_i as C_{sig}.
8. Create the tracing tag tt_s as $(mid, C_K, C_{PK}, ts, C_{Sig})$, to be delivered to the server when sending the associated message. (The tracing key k_i is encrypted with the message, to be delivered to the recipient only.)

Svr-Process$(DB, U_r, tt_s) \rightarrow ((mid, tt_p), tt_r)$**:**

1. Check the database DB for an existing entry under *mid*. If one exists, reject the message.
2. Check that the timestamp is recent, if not reject the message.
3. Add the current tracing information to DB, with *mid* as the key value.
4. Copy tt_s to create the tracing tag for the recipient, tt_r.

RecMsg$(k_i, cert_{U_s}, U_r, m, tt_r) \rightarrow k_i$**:**

1. Verify that the *mid* in tt_r can be generated using the received message m and tracing key k_i with the PRF. If not reject the message.
2. Decrypt C_{PK} and verify that the public key $LTPK_{U_S}$ matches the sending user's certificate, $cert_{U_s}$. If not reject the message.
3. Decrypt and verify the signature within C_{sig} using the information from tt_r, if verification fails reject the message.
4. If all verification succeeds, display the message. Output k_i can be used to report the message for tracing.

Svr-Trace$(DB, U, m, k) \rightarrow Tr$**:**

1. Initialize a list Tr of tracing information, beginning with the reporter.

2. Use the supplied m and k to generate $mid = \mathsf{F}_{\tilde{k}}(m)$.
3. Look up the generated mid within DB, and retrieve the relevant tracing information. If the lookup fails, the trace has concluded.
4. Decrypt C_{PK} to identify the sender with $LTPK_{U_s}$.
5. Decrypt sig from C_{sig} and verify, if it fails the trace has concluded.
6. Add the user's identity U_s and message ID mid to Tr.
7. Decrypt C_K to retrieve the next tracing key, replacing k, and calculate a new mid using the new k and the reported message m.
8. Repeat steps 3–7 until a lookup fails (indicating the original sender) or a signature fails to verify (indicating a bad signature was knowingly accepted). Tr will now contain the user identities and associated mid values (for reference) in reverse order.

Security Analysis. Our Anonymous Path Traceback construction achieves trace confidentiality, anonymous trace unforgeability, and pre-trace anonymity for both originators and forwarders of a given message. Proof sketches are in the appendix, the complete proofs are left to the full version [4].

4 Anonymous Source Traceback

Path traceback is inherently limited when our goals include maintaining as much anonymity as possible within the system, so with the basics ironed out we turn towards preserving the anonymity of the links along the chain. This is called source traceback, which reveals *only* the original source's identity.

The primary obstacle to achieving source traceback based on our existing construction is the structure of our database itself. As we follow the trail of message IDs, at each and every step we verify the signature at that entry, revealing the identity of the person who sent or forwarded it. This is unavoidable with a step-by-step method and identity information present.

We solve this by splitting the server in half, passing the identity-related information to a message server that delivers the messages and giving necessary tracing information to a tracing server which follows the chain of mid evaluations and lookups to its end. When tracing, the tracing server passes its result along to the message server, who can then verify the signature and learn the identity. Splitting the database is the only option to continue with the current structure of the system, where traces are made one step at a time. There has to be information in the trace capable of revealing the user's identity, and no real way to prevent the platform from recovering it when performing a trace, since it is necessary to find the source.

Technical Challenges: There are challenges in building an *extremely simple* two-party protocol for source traceback in our split server model. The main challenge is how to deal with a failed signature verification. The message server can only verify the signature of the final result, so what happens if that fails? It must go back to the tracing server for more information. So the tracing server must retain information on its most recent traces and respond to the message server

when it needs help. However, this creates another problem: what is stopping the message server from simply asking for all the information whether they need it or not, reverting to path traceback? Assuming the tracing server is unwilling to collude with the message server, we can solve this issue by introducing an extra signature. This signature, made using ephemeral one-use keys to sign the tracing data entry that the message server handles, can be used to prove to the tracing server that the failed signature justifying its request is legitimate.

Additionally, the two servers need to ensure that the tracing information for any message is present on both servers; the opposite half of the tracing information must be guaranteed to exist on the other server. This manifests in the form of a small amount of communication to allow the message server to reject messages that have no corresponding information in the tracing server. Aside from this, the split also creates a theoretical issue; no longer can the encryption of the previous key, C_K, be verified, which opened a few interesting questions that are answered in the full version's [4] security proof.

In this way we can maintain the efficiency and the simplicity of the first construction while achieving anonymity for all but the reporter and the "source", if the underlying messaging system's anonymity is strong enough. With only sender anonymity as mentioned before we leak only the first recipient's identity.

Benefits and Instantiations of Split Servers: The split server construction does have benefits of its own as well. For example, prior to this construction a user colluding with someone who has access to the server's database could reveal an entire forwarding path; in essence making a report that will definitely result in a trace. With the message server and tracing server as distinct entities this collusion becomes more difficult. Access to just the tracing server's database cannot reveal identities, while the message server's database would reveal the identity of the immediate sender, but tracing to the originator would be impossible. In addition, we can allow more privacy to originators: the tracing server learns only the message, not their identity, and the message server learns only the identity without the message. Passing the message to the message server is a trivial change that does not affect any of the security properties, but it is an extra layer of privacy available due to the split.

As for instantiation, there are several possible options for the two non-colluding servers needed to maintain our anonymity properties, including trusted hardware on the messaging server, or having one server run by an independent entity such as an NGO or non-profit. We also remark that the other relevant solution, Hecate [6], implicitly requires a similar structure; while their moderator and server can technically be separate, doing so would reduce their anonymity guarantee for token-less originators similar to [7].

4.1 Construction Details

Compared to the anonymous path traceback construction, while we add an amount of communication overhead, the primary effect is splitting the server-side work between the message server and the tracing server.

Where originally a database entry contained a mid as a key pointing to $C_K, C_{cert}, C_{sig}, k_T$, and the recipient's identity, these are now split. The tracing server's database contains only the information necessary to follow a chain of entries, mid and C_K, while as the message server delivers the message to the recipient, its database carries information related to the signature, $mid, C_{cert}, C_{sig}, ts$, and k_T. In addition to splitting the old information, both servers store the new ephemeral public key, pk_{eph}, and the message server stores the signature it verifies, sig_{eph}. One new wrinkle is that, as the message server's database no longer contains C_K, it can no longer be signed on by the signature. This will be covered in more detail in the security proofs, but during analysis it will be shown that C_K actually does not need to be verified by the signature.

The client-side algorithms contain only minor modifications:

TagGen$(U_s, m, k_prev, LTSK_{U_s}) \rightarrow (k_i, tt_{ms}, tt_{ts}, sig)$**:**

1. Follow steps 1–7 from the Anonymous Path Traceback version of TagGen.
2. Generate an ephemeral asymmetric key pair pk_{eph} and sk_{eph}.
3. Generate sig_{eph} by signing on $(mid, C_{PK}, ts, C_{sig})$ with sk_{eph}.
4. Create the message server's tracing tag tt_{ms} as $(mid, C_{PK}, ts, C_{Sig}, pk_{eph}, sig_{eph})$, and the tracing server's tracing tag tt_{ts} as (mid, C_K, pk_{eph}).

RecMsg$(k_i, cert_{U_s}, U_r, m, tt_r) \rightarrow k_i$**:**

1. Act as Anonymous Path Traceback's RecMsg. The only difference is removing C_K from the signature verification.

The server-side algorithms however, have split into two, and add **TSvr-Req**, to handle the message server's requests for additional information:

TSvr-Process$(TDB, tt_{ts}) \rightarrow mid$**:**

1. Check the database TDB for an existing entry under mid. If one exists, reject the message.
2. Add the information within tt_{ts} to TDB, with mid as the key.
3. Pass mid along to the message server to notify that the tracing server's half of the tracing data has been received for that message.

MSvr-Process(MDB, U_r, tt_{ms})**:**

1. Check the database MDB for an existing entry under mid. If one exists, reject the message.
2. Check that the timestamp ts is recent. If not reject the message.
3. Verify sig_{eph}, reject the message if this fails.
4. Add the information within tt_{ms} to MDB, with mid as the key.
5. Create the tracing tag tt_r as $(mid, C_{PK}, ts, C_{Sig})$, for the recipient.

TSvr-Trace$(TDB, TLOG, m, k) \rightarrow (mid, k, tid)$**:**

1. Initialize a list Tr of tracing information.
2. Use the supplied m and k to generate $mid = \mathsf{F}_{\tilde{k}}(m)$.
3. Look up the generated mid within TDB, and retrieve the tracing information. If the lookup fails, the trace has concluded.
4. Add the mid and associated key k to Tr.
5. Decrypt C_K to retrieve the next tracing key, replacing k, and calculate a new mid using the new k and the reported message m.

6. Repeat steps 3–5 until a lookup fails. Tr will now contain the *mid* values and associated keys of the message chain in reverse order.
7. Store the full trace information Tr in $TLOG$ to be referenced later if needed, generating a trace ID tid as the key.
8. Return the end result mid and key to the message server along with tid.

MSvr-Trace$(MDB, mid, k, tid) \rightarrow U_s$**:**

1. Look up the provided mid value in MDB, then decrypt sig from the associated C_{sig} and verify.
2. If the signature fails to verify, request the next mid in the trace from the tracing server using TSvr-Req, and repeat step 1.
3. When the signature verifies, either the original source or a user who purposely accepted a bad signature has been identified.

TSvr-Req$(TDB, TLOG, mid, k, tid, tt_{ms}) \rightarrow (mid, k_{prev}, tid)$**:**

1. Look up the provided mid value in TDB and verify that the provided tt_{ms} matches, sig_{eph} verifies, and sig within C_{sig} does not verify to prove the message server requires additional information. If this fails, reject the request.
2. Look up the provided tid in $TLOG$, and return the next mid value in the trace to the message server.

Security Analysis. Our Anonymous Source Traceback construction achieves trace confidentiality from both user and platform, anonymous trace unforgeability, pre-trace anonymity for message originators, and post-trace anonymity for message forwarders; leaving forwarders unknown even after a successful trace of a message they forwarded. For further details and proof sketches see the appendix, the complete proofs are left to the full version [4].

5 Implementation and Performance

Our implementation focused on testing the cryptographic overhead our constructions cause on the clients and servers of an E2EE system. The implementation was programmed in C, using the libsodium cryptographic libary. While there does exist an implementation of the original traceback [13], one of its dependencies no longer exists and the successor has a different API, rendering the original implementation immeasurable, so we started from scratch. For the hash function H and PRF F, we use the Blake2b algorithm, encryption uses the XChaCha20 algorithm, for signatures we used EdDSA. We also make use of the libsignal-client general purpose Signal library as a point of comparison; its session and sealed sender benchmarks can estimate the overhead of the double-ratchet and sealed sender. All tests were run on an Intel i7-11800H processor with turbo disabled, meaning the processor was locked to its base clock speed of 2.3 Ghz.

Table 2. Average cryptographic overhead for client-side. In order: baseline Signal overhead, then additional overhead from Sealed Sender, then further additional overhead from [13] or from our own constructions. Compared to Hecate [6], who tested with a stronger processor, our anonymous source traceback construction's sending overhead is comparable to their combined token generation and sending overhead, with our recipient overhead much lower.

	Signal	Sealed sender	[13]	Anon path	Anon source
Sending (μs)	7.0958	198.69	+1.569	+34.7775	+96.5644
Receiving (μs)	204.15	174.42	+0.6075	+80.992	+80.992

Performance Results. Looking at the client-side overhead in Table 2, there is obviously an overhead with our constructions compared to the non-anonymous traceback [13], however even in the worst case we remain under 1 ms on average. Compared to the overhead Signal users already face, this is definitely feasible.

Server-side processing overhead is workable as well. In our anonymous path traceback construction we use no additional cryptography, so we only see an increase in overhead when verifying the ephemeral key signature in our anonymous source traceback construction, which costs on average 79.82 µs. While this may sound like a large performance hit compared to no cryptographic overhead at all, on average it can be calculated 12,528 times in a second. Given that we are testing with a single thread on a CPU locked to its base clock speed, in a data center this number will scale to a much higher figure.

Overhead during tracing is also worth discussing. In anonymous path traceback every signature in a trace must be verified so overhead scales directly with the length of the trace (approx. 81.919463 µs per forward). In anonymous source traceback only the original's overhead occurs per forward (approx. 1.338296 µs) while signature verifications are conditional; only if the message server's initial verification fails must the tracing server and message server perform further verifications, costing approximately 80.552661 µs to the message server and 160.840751 µs to the tracing server per bad signature. Most traces will not contain many bad signatures, so on average there should be much less overhead in anonymous path traceback. Though it is worth considering the possibility of purposefully chaining garbage signatures as a denial of service attack.

To examine our space efficiency we use the same estimation as [13]: a messaging service that uses a 1-month window and sees 1 billion messages per day. In anonymous path traceback a database entry is 136 bytes, leading to a requirement of ≈4.08 TB, while for anonymous source traceback entries in the tracing database are 96 bytes, requiring ≈2.88 TB, and message database entries are 232 bytes, requiring ≈6.96 TB. Compared to the original's figures [13] of ≈600 GB for path traceback and ≈2 TB for tree traceback, we do take up quite a bit more space. However, while the results are not exciting, they are still likely feasible for a large service, and our implementation uses conservatively high key sizes, so there's room to reduce while maintaining security.

Implementation Concerns. One thing the original [13] leaves unexamined is the sliding window approach they recommend for databases when implementing their system. This puts a cap on storage costs, but it creates a surprising number of problems in the process. Previously discussed in more detail is the delayed replay attack, though that is only an issue due to anonymity.

A general issue for traceback-style systems that use a sliding window is behavior when a message trace is called for after the original source has timed out of the database. A lookup failure indicates that the original source has been found, but if lookup fails because an entry no longer exists, the trace could penalize the wrong person. Using a sliding window thus requires care. At the very least, a 2-part window that first flags an entry as timed out, then deletes after a second time window would avoid misattribution of blame, though the trace still would not find the source.

It is worth considering preventing forwards client-side after the window has passed, but this may do more harm than good. The likelihood of a user just copy-pasting is high, and the alternative of a confirmation prompt warning that they will be held responsible may not prevent it. Overall, the limitations of the sliding window should be kept in mind.

Acknowledgements. The authors were partially supported by NSF CNS #1801491. Qiang is also partially supported by gifts from Ethereum Foundation, Protocol Labs, Stellar Foundation, and Algorand Foundation.

6 Proof Sketches

6.1 Anonymous Path Traceback

Trace Confidentiality. For *trace confidentiality*, both user and platform, we can refer back to [13] for their original proofs, as we add no additional information that could be used to distinguish forwarded messages from original messages. This is fairly straightforward to see, while we add (depending on the system), a Signature, Timestamp, and/or Anonymous Blacklisting Authentication Token, these components do not vary between original or forwarded messages. The only difference remains the value of the previous key encrypted as C_K, just as it was in the original Traceback paper, and so we can inherit their security here.

Theorem 1. *With APT as the anonymous path traceback scheme defined in Sect. 3.1: For any AnonTrUNF adversary $\mathcal{A}$, there are corresponding adversaries $\mathcal{B}$ and $\mathcal{C}$ running in the same time as $\mathcal{A}$ such that:*

$$\mathbf{Adv}_{APT}^{AnonTrUNF}(\mathcal{A}) \leq \mathbf{Adv}_{F}^{cr}(\mathcal{B}) + \mathbf{Adv}_{Sig}^{forge}(\mathcal{C})$$

For any PreAnon adversary $\mathcal{A}$, there is a corresponding adversary $\mathcal{B}$ running in the same time as $\mathcal{A}$ such that:

$$\mathbf{Adv}_{APT}^{PreAnon}(\mathcal{A}) \leq \mathbf{Adv}_{ENC}^{cpa}(\mathcal{B})$$

Trace Unforgeability. As seen in Theorem 1, the adversary's advantage against *anonymous trace unforgeability* is a sum of the advantage against the PRF's collision resistance and the advantage for forging a signature. This means their advantage should be negligible, as otherwise the probability of breaking one of the two secure building block schemes would be non-negligible.

Proof Sketch. For *anonymous trace unforgeability*, the same four failure cases still form the basis of the proof:

- Case 1: An empty trace.
- Case 2: The identified honest original sender never sent the message.
- Case 3: The reporter never received the message they reported.
- Case 4: An honest user identified as a forwarder did not forward the message.

However, we must also account for the adversary's additional capabilities; specifically, it is no longer guaranteed that the identity stored matches the identity of the user who actually sent the message. To model this the **SendMal** Oracle now allows much more freedom to the adversary. In addition, we model the possibility of a change in the sliding window with the **ClearDB** oracle.

To account for the adversary's new capabilities we add a game transition; **SendMal** sets *BadSend* when the sender identity does not match the tag. This separates out the situations where the original traceback security proof's assumptions fail. Regardless of why the identities do not match, for the message to have been accepted means a signature must have been forged.

The remaining failure cases are handled as they are in the original [13] proof, with one exception. Cases 1 and 3 are impossible because they require an honest user to report a message they never received; the **Send** and **SendMal** oracles both set WasRec. Case 2, U_j falsely identified as original source for a message they did not send, cannot happen in absence of signature forgeries due to PRF collision resistance: a trace for a different plaintext or key must result in the same *mid* as a different message U_j sent.

Case 4 is similar to Case 2, U_j is falsely identified as a forwarder, and in absence of signature forgery this also requires a PRF collision. Either in the exact same manner as Case 2, or in a special case where U_j is actually the original source. In the original proof this is designated "problematic" and several game transitions are used to isolate it, but on second inspection this is still the result of a PRF collision between the plaintext and fake k_{prev} with some unrelated message and key.

Pre-trace Anonymity. Anonymous path traceback aims for *pre-trace anonymity* for both the originators and forwarders of its messages, therefore in both cases the **Send** oracle tracks forwards of the challenge message and **Trace** oracles disallow tracing those messages.

Proof Sketch. For both *OriginAnon* and *ForwarderAnon* the proofs are nearly identical. In both cases security is guaranteed by encryption; the only useful information the adversary has is the server's view of the challenge messages.

When looking at that view, all relevant information is encrypted, so breaking encryption is necessary to learn the sender's identity. There is one extra wrinkle for *ForwarderAnon*; the adversary can choose the key that will be encrypted by U_b as their C_K value, which the adversary has access to through the **DB** oracle. However, attempting to recover the key in this way corresponds to a chosen plaintext attack, which would also break the encryption's security.

6.2 Anonymous Source Traceback

Theorem 2. *With AST as the anonymous source traceback scheme defined in Sect. 4.1: For any AnonTrUNF adversary $\mathcal{A}$, there are corresponding adversaries $\mathcal{B}$ and $\mathcal{C}$ running in the same time as $\mathcal{A}$ such that:*

$$\mathbf{Adv}_{AST}^{AnonTrUNF}(\mathcal{A}) \leq \mathbf{Adv}_{F}^{cr}(\mathcal{B}) + \mathbf{Adv}_{Sig}^{forge}(\mathcal{C})$$

For any FPostAnon adversary $\mathcal{A}$, there are corresponding adversaries $\mathcal{B}$ and $\mathcal{C}$ running in the same time as $\mathcal{A}$ such that:

$$\mathbf{Adv}_{AST}^{FPostAnon}(\mathcal{A}) \leq \mathbf{Adv}_{ENC}^{cpa}(\mathcal{B}) + \mathbf{Adv}_{Sig}^{forge}(\mathcal{C})$$

Originator Pre-trace Anonymity. This aims for *pre-trace anonymity* for originators, so as in path traceback, the **Send** oracle tracks forwards of the challenge message and **Trace** oracles disallow tracing those messages.

Proof Sketch. As the amount of information in the hands of the adversary has slightly shrunk as compared to anonymous path traceback, things remain largely the same as the previous *pre-trace anonymity* proof.

Just as before, the only useful things here are C_{PK} and C_{sig}, which must be decrypted to utilize, and whose key is unavailable and generated independent of any other information. Therefore, breaking the encryption remains necessary.

Theorem 3. *With AST as the Anonymous Source Traceback scheme defined in Sect. 4.1, for any OPreAnon adversary $\mathcal{A}$, there is a corresponding adversary $\mathcal{B}$ running in the same time as A such that:*

$$\mathbf{Adv}_{AST}^{OPreAnon}(\mathcal{A}) \leq \mathbf{Adv}_{ENC}^{cpa}(\mathcal{B})$$

Forwarder Post-trace Anonymity. Anonymous source traceback aims to give forwarders *post-trace anonymity*, so unlike the previous anonymity definitions, the **Send** and **Trace** oracles do not limit tracing in any way. However, to account for the new avenue the message server has in gathering information, we also add the **Request** oracle to allow querying the tracing server.

Proof Sketch. The new **Request** oracle allows the adversary to attempt to gain information on forwarders of a message after a trace is complete, however to do so would require forging the ephemerally keyed signature meant to ensure the Message Server's honesty. We use a game transition to isolate this possibility. Outside of that new possibility, the adversary cannot learn path information from the Tracing Server. While we now allow tracing messages downstream from the forwarder whose identity we want to protect, this gives no real advantage without the path information, so breaking encryption is still required to learn the forwarder's identity.

Anonymous Trace Unforgeability. The primary difference from the anonymous path traceback proof is that no tracing information can be verified at the time of tracing aside from the final result's. For most of that information, there is no real benefit to providing bad entries; the signature will fail to verify and honest recipients will drop the message. The one interesting case is C_K, which is no longer included in the signature. If C_K could be chosen properly, it would redirect a trace in a completely different direction, but that still requires violating the collision resistance property of the PRF. As we no longer have to worry about the full message path remaining accurate, only two failure conditions remain: an empty trace, and a misidentified source. These reduce in the same way as the previous unforgeability proof; if the identities mismatch a signature was forged, and otherwise a PRF collision occurred.

References

1. Bond, S.: Just 12 people are behind most vaccine hoaxes on social media, research shows. NPR (2021). https://www.npr.org/2021/05/13/996570855/disinformation-dozen-test-facebooks-twitters-ability-to-curb-vaccine-hoaxes
2. For Countering Digital Hate, C.: The disinformation dozen (2021). https://www.counterhate.com/disinformationdozen
3. Dodis, Y., Grubbs, P., Ristenpart, T., Woodage, J.: Fast message franking: from invisible salamanders to encryptment. Cryptology ePrint Archive, Report 2019/016 (2019). https://ia.cr/2019/016
4. Anonymous traceback for end to end encryption. https://drive.google.com/file/d/1uDBndw3dvAK2Ep_ocwovSzabPl1wXLrT/view?usp=sharing
5. Grubbs, P., Lu, J., Ristenpart, T.: Message franking via committing authenticated encryption. Cryptology ePrint Archive, Report 2017/664 (2017). https://ia.cr/2017/664
6. Issa, R., AlHaddad, N., Varia, M.: Hecate: abuse reporting in secure messengers with sealed sender. Cryptology ePrint Archive, Report 2021/1686 (2021). https://ia.cr/2021/1686
7. Liu, L., Roche, D.S., Theriault, A., Yerukhimovich, A.: Fighting fake news in encrypted messaging with the fuzzy anonymous complaint tally system (facts). Cryptology ePrint Archive, Report 2021/1148 (2021). https://ia.cr/2021/1148

8. Peale, C., Eskandarian, S., Boneh, D.: Secure complaint-enabled source-tracking for encrypted messaging. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS 2021, pp. 1484–1506. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460120.3484539
9. Samuels, E.: How misinformation on whatsapp led to a mob killing in India. The Washington Post (2020). https://www.washingtonpost.com/politics/2020/02/21/how-misinformation-whatsapp-led-deathly-mob-lynching-india/
10. Government requests. https://signal.org/bigbrother/
11. Tyagi, N., Grubbs, P., Len, J., Miers, I., Ristenpart, T.: Asymmetric message franking: content moderation for metadata-private end-to-end encryption. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11694, pp. 222–250. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26954-8_8
12. Tyagi, N., Len, J., Miers, I., Ristenpart, T.: Orca: blocklisting in sender-anonymous messaging. Cryptology ePrint Archive, Report 2021/1380 (2021). https://ia.cr/2021/1380
13. Tyagi, N., Miers, I., Ristenpart, T.: Traceback for end-to-end encrypted messaging. Cryptology ePrint Archive, Report 2019/981 (2019). https://ia.cr/2019/981
14. Vasilogambros, M.: Disinformation may be the new normal, election officials fear. PEW (2021). https://www.pewtrusts.org/en/research-and-analysis/blogs/stateline/2021/09/21/disinformation-may-be-the-new-normal-election-officials-fear